CHAPITRE 12 : LES MODULES :NUMPY, MATPLOTLIB, SCIPY

En Python ils existent deux façons de déclarer des tableaux (matrices) :
- comme des listes
- comme des tableaux mathématiques en utilisant la bibliothèque **numpy**.

Dans ce chapitre nous allons utiliser la deuxième approche.

I. Le module numpy :

I-1. Représentation d'un tableau (matrice) par le module numpy :

Les listes sont très utilisées et indispensables pour la maîtrise de Python. Cependant, elles ont quelques limitations qui les rendent plus difficiles à utiliser dans le contexte mathématique. Par exemple, opération * (multiplication) ne correspond pas à la multiplication d'un vecteur avec un nombre.
La bibliothèque **numpy** a été crée pour résoudre ce problème. En plus d'un grand nombre de fonctions, elle défini un autre type de tableaux, spécialement conçus pour les opérations mathématiques, tels que vecteurs, matrices et tenseurs.

I-2. Importer le module numpy

Pour utiliser le module **numpy** il faut commencer par l'importer :

```
import numpy as np
```

Dorénavant, pour accéder aux fonctions définies par numpy nous devons ajouter le préfixe np,

Exemple :

x=np.sin(3) print(x)	le script affiche : **0.14112000806**

Si on n'a besoin que d'une seule fonction : **from numpy import sin**

I-3. Quelques fonctions mathématiques présentes dans numpy :

La fonction	Description	Exemple	Le script affiche
np.pi	la constante π	print(np.pi)	3.141592653589793
np.abs()	la valeur absolue	print(np.abs(-23)) print(np.abs(1+1j))	23 1.41421356237
np.rad2deg() **np.deg2rad()**	conversion de radians en dégrées et vice versa. **Notez bien, que partout en Python les angles sont donnés en radians.**	print(np.deg2rad(180)) print(np.rad2deg(np.pi))	3.14159265359 180.0
np.cos() **np.sin()** **np.tan()**	les fonctions trigonométriques de base.	print(np.cos(np.pi)) print(np.sin(np.deg2rad(45))) print(np.tan(np.pi/4))	-1.0 0.707106781187 1.0

np.arccos() **np.arcsin()** **np.arctan()**	les fonctions inverses de cos, sin et tan	x= np.cos(np.pi) print(np.arccos(x))	3.14159265359
np.ceil()	arrondi au plus petit entier supérieur	print(np.ceil(3.01))	4.0
np.floor()	arrondi au plus grand entier inférieur	print(np.floor(3.99))	3.0
np.round()	arrondi au plus proche	print(np.round(3.49)) print(np.round(3.50))	3.0 4.0

Pour voir la liste complète de fonctions, utilisez la fonction **dir** :

- **dir()** : affiche la liste des variables actuellement définies
- **dir(np)** : # Affiche la liste avec toutes les fonctions définies par np

I-4. Créer un tableau array :

numpy ajoute le type **array** qui est similaire à une liste (list) avec la condition supplémentaire que tous les éléments sont du même type. Ils existent quelques moyens pour la définition d'un tableau numpy :

✓ **A partir de listes :**

# pour un tableau à une dimension 	
M = **np.array**(liste)
	L=**[1,2,5,20]**
	M=**np.array(L)**
	print(M)
		Le script affiche :
		[1 2 5 20]

# pour une matrice : chaque ligne #étant une liste 	
M = **np.array**([ligne1, ligne2, ...])
	L1=[1,2,5]
	L2=[10,-2,3]
	L3=[10,-2,3]
	M=**np.array([L1,L2,L3])**
	print(M)
		Le script affiche :
		[[1 2 5]
		[10 -2 3]
		[10 -2 3]]

	M=**np.array([[0]*4 for i in range(3)])** print(M)	Le script affiche : **[[0 0 0 0]** **[0 0 0 0]** **[0 0 0 0]]**

#On peut préciser le type des #éléments du tableau, M = **np.array**(liste, dtype = type) # type : peut-être int, float, #complex, ... #mais attention, tous les éléments #du tableau doivent avoir le même #type	L=[1,2,5] M1=**np.array(L,dtype=float)** print('M1=',M1) M2=**np.array(L,dtype=complex)** print('M2=',M2)	Le script affiche : **M1= [1. 2. 5.]** **M2= [1.+0.j 2.+0.j 5.+0.j]**

✓ **A partir de : arange, linspace**

M = **arange**(début, fin, pas) **#** la valeur « fin » est non comprise , #résultat = tableau unidimensionnel #. Le pas peut-être un flottant (ce #qui n'est pas le cas pour range).	M=**np.arange(0,5,1.2)** print(M)	Le script affiche : **[0. 1.2 2.4 3.6 4.8]**

M = **linspace**(début, fin, nb valeurs) **#** retourne un tableau de points #régulièrement espacés entre « #début » et « fin ». La valeur « fin » #est comprise	M=**np.linspace(0,10,3)** print(M)	Le script affiche : **[0. 5. 10.]**

✓ **Matrices aléatoires :**

M = **np.random.rand**(n, p) #matrice de nombres aléatoires #entre 0 et 1 avec n lignes et p #colonnes (loi uniforme)	M = **np.random.rand**(4, 2) print(M)	Le script affiche : **[[0.76411009 0.28516386]** **[0.03382596 0.7617002]** **[0.0789823 0.15047101]]**

✓ **Matrices particulières :**

#matrice diagonale
M = **np.diag**(liste)
	M = **np.diag(**[1,2,5])
	print(M)
		Le script affiche :
		[[1 0 0]
		[0 2 0]
		[0 0 5]]

#matrice avec k décalage par
#rapport à la diagonale principale
M = **np.diag**(liste, k)
	M = **np.diag(**[1,2,5],**1)**
	print(M)
		Le script affiche :
		[[0 1 0 0]
		[0 0 2 0]
		[0 0 0 5]
		[0 0 0 0]]

# matrice nulle de taille (n, p) 	
#1 (attention #aux double
#parenthèses)
M = **np.zeros**((n, p))
	M = **np.zeros((2,4))**
	print(M)
		Le script affiche :
		[[0. 0. 0. 0.]
		[0. 0. 0. 0.]]

#matrice dont toutes les #composantes sont des 1 (attention #aux double-parenthèses)	M = **np.ones((2,4))** print(M)	Le script affiche : **[[1. 1. 1. 1.]** **[1. 1. 1. 1.]]**

M = **ones**((n, p))		

#matrice identité de taille n
M = np.**eye**(n)
	M = **np.eye(3)**
	print(M)
		Le script affiche :
		[[1. 0. 0.]
		[0. 1. 0.]
		[0. 0. 1.]]

I-5. Informations sur un tableau

Pour avoir les informations sur un tableau numpy :

#Nombre d'éléments d'un tableau **mon_tableau.size**	T=np.array([1,-2,5]) print(**T.size**) M= np.array([[1,-2,5],[0,1,2]]) print(**M.size**)	Le script affiche : **3** **6**

# Dimensions du tableau **mon_tableau.shape**	T=np.array([1,-2,5]) M= np.array([[1,-2,5],[0,1,2]]) # pour un vecteur (tableau #unidimensionnel),on obtient (n,) **x=T.shape** # pour une matrice, on obtient #(nb lignes, nb colonnes) **y=M.shape** print(x) print(y)	Le script affiche : **(3,)** **(2, 3)**

# Type des éléments du tableau (le #même pour tous les éléments) **mon_tableau.dtype**	T1=np.array([1,-2,5]) T2=np.array([1.5,-2,5]) print(**T1.dtype**) print(**T2.dtype**)	Le script affiche : **int32** **float64**

I-6. Lire/Extraire des données d'un tableau (tranches) :

✓ <u>lecture (cas des tableaux unidimensionnels = vecteurs) :</u>

indice de ligne	0	1	2	...		...	− 2	− 1
valeur	v[0]	v[1]	v[2]	...		...	v[−2]	v[−1]

v[k]	valeur de la case k, (les indices commencent à 0 comme pour les listes).
v[i : j]	lecture par tranche (indice j non compris) ; syntaxe **v[start :stop :step].**
v[i :]	toutes les composantes à partir de celle d'indice i incluse.
v[: j]	toutes les composantes jusqu'à celle d'indice j non comprise.
v[- k]	lecture de la k-ième composante à partir de la fin
v[-k :]	lecture des k derniers éléments.

v[: : 2]	composantes lues de 2 en 2.

✓ <u>lecture (cas des tableaux bidimensionnels = matrices) :</u>

	0	1	2				-2	-1
0	**M[0,0]**	**M[0,1]**	**M[0,2]**				**M[0,-2]**	**M[0,-1]**
1	**M[1,0]**							**M[1,-1]**
2	**M[2,0]**							**M[2,-1]**
-2	**M[-2,0]**							**M[-2,-1]**
-1	**M[-1,0]**	**M[-1,1]**	**M[-1,2]**				**M[-1,-2]**	**M[-1,-1]**

M[i,j] : représente la valeur de la case d'indice (i,j) d'un tableau bidimensionnel, Les indices démarrent à 0.
Noter la différence avec les listes de listes pour lesquelles on doit écrire obligatoirement M[i][j].

M[i]	i-ième ligne de M qui est un tableau à une dimension ou bien **M[i, :]**
M[– i]	i-ième ligne de M à partir de la fin.
M[:, j]	j-ième colonne de M.
M[i1 :i2]	lignes de M, de celle d'indice i1 comprise à celle d'indice i2 non comprise.
M[:, j1 :j2]	colonnes de M, de celle d'indice j1 comprise à celle d'indice j2 non comprise.
M[i1:i2 , j1:j2]	lignes i1 incluse à i2 exclue et colonnes j1 incluse à j2 exclue.
M[i :]	toutes les lignes de M à partir de la ligne d'indice i incluse.
M[: i]	toutes les lignes de M jusqu'à la ligne d'indice i non comprise ;
M[: , :j]	toutes les colonnes de M jusqu'à celle d'indice j exclue.
M[::2]	lignes de M de 2 en 2
M[[i1,i2,...]]	lignes d'indices i1, i2, … On obtient un tableau bidimensionnel même s'il n'y a qu'une seule ligne.
M[X,Y]	X et Y des listes d'entiers : vecteur [M[x1,y1] , M[x2,y2], ...].

I-7. Modifications d'un tableau :

✓ <u>Changer le format d'un tableau :</u>

M1 = M.reshape(n, p)	M=np.arange(8) M1=**M.reshape(2,4)** print("M=",M) print("M1=",M1)	Le script affiche : **M= [0 1 2 3 4 5 6 7]** **M1= [[0 1 2 3]** **[4 5 6 7]]**

<u>Rq :</u> le tableau reformaté M1 pointe à la même adresse que M, donc toute modification de M se répercute à M1 et vice-versa.
Il faut que le nombre de composantes de M soit égal à **n*p**.

✓ Modifier la (les) valeur(s) d'un tableau:

#tableau unidimensionnel #Modification d'un seul terme : **v[k] = valeur**	v=np.linspace(0,10,5) print(v) **v[0]=200** print(v)	Le script affiche : **[0. 2.5 5. 7.5 10.]** **[200. 2.5 5. 7.5 10.]**
#tableau bidimensionnel #Modification d'un seul terme : **M[i,j] = valeur**	M=np.ones((2,3)) print('Avant\n',M) **M[0,2]=200** print('Après\n',M)	Le script affiche : **Avant** **[[1. 1. 1.]** **[1. 1. 1.]]** **Après** **[[1. 1. 200.]** **[1. 1. 1.]]**

#tableau bidimensionnel #Modifications simultanées : **M[1, :] = valeur** #tous les termes de la ligne d'indice #1 sont mis à **valeur**	M=np.ones((2,3)) print('Avant\n',M) **M[1, :]=0** print('Après\n',M)	Le script affiche : **Avant** **[[1. 1. 1.]** **[1. 1. 1.]]** **Après** **[[1. 1. 1.]** **[0. 0. 0.]]**
M[: ,1] = valeur #tous les termes de la colonne #d'indice 1 sont mis à **valeur**	M=np.ones((2,3)) print('Avant\n',M) **M[:, 0]=0** print('Après\n',M)	Le script affiche : **Avant** **[[1. 1. 1.]** **[1. 1. 1.]]** **Après** **[[0. 1. 1.]** **[0. 1. 1.]]**
M[[i1,i2, ...]] = valeur #remplir simultanément plusieurs lignes désignées par leurs indices i1,i2,.. par la valeur **valeur**.	M=np.ones((4,3)) print('Avant\n',M) **M[[0,2]]=4** print('Après\n',M)	Le script affiche : **Avant** **[[1. 1. 1.]** **[1. 1. 1.]** **[1. 1. 1.]** **[1. 1. 1.]]** **Après** **[[4. 4. 4.]** **[1. 1. 1.]** **[4. 4. 4.]** **[1. 1. 1.]]**
M[: ,[i1,i2, ...]] = valeur #remplir simultanément plusieurs colonnes désignées par leurs indices i1,i2,.. par la valeur **valeur**.	M=np.ones((4,3)) print('Avant\n',M) **M[: ,[0,2]]=4** print('Après\n',M)	Le script affiche : **Avant** **[[1. 1. 1.]** **[1. 1. 1.]** **[1. 1. 1.]** **[1. 1. 1.]]** **Après** **[[4. 1. 4.]** **[4. 1. 4.]** **[4. 1. 4.]** **[4. 1. 4.]]**

I-8. copie d'un tableau :

Comme pour les listes, une assignation M1 = M ne crée pas une vraie copie de M. M1 est juste un alias de M et pointe à la même adresse. Pour obtenir une vraie copie, on a recours à la méthode copy :

M1=M.copy() #ou bien **M1 = np.copy(M)**

Exemple :

```
M = np.zeros((3,4))
M1 = M.copy()
print('Avant la modification :')
print('M :',M)
print('M1 :',M1)
M[0:,0]=13
print('Après la modification :')
print('M :',M)
print('M1 :',M1)
```

Le script affiche :

```
Avant la modification :
M : [[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
M1 : [[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
Après la modification :
M : [[ 13.   0.   0.   0.]
 [ 13.   0.   0.   0.]
 [ 13.   0.   0.   0.]]
M1 : [[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

I-9. Calcul numérique matriciel :

Les opérations + et * sont des opérations terme à terme sur les tableaux (**rappel : ce n'est pas le cas pour les listes**).

- **k*v** , **k*M** : toutes les composantes du vecteur v (resp. la matrice M) sont multipliées par k
- **v+k** , **M + k** : on rajoute k à toutes les composantes du vecteur v (resp. la matrice M).
- **v*v** : produit terme à terme (carré de la norme du vecteur v)
- **M*M** : produit terme à terme

  Attention !!

  Le produit matriciel habituel est obtenu par la fonction **dot**:

  M.dot(M) ou **np.dot(M,M)** #le résultat est un array

- **M.T** : transposée de la matrice M . toute modification sur **M.T** se répercute sur M.
- **np.sum(M)** : somme des termes de M
- **Les tests sur les matrices** :

```
M1 = = M2
#comparaison de toutes
#les composantes de M1
#avec M2 #terme à terme
```

```
M1=np.array([1,-2,5,9,51])
M2=np.array([1,-2,5,51,9])
x=M1==M2
print(x)
```

Le script affiche :

```
[ True  True  True False False]
```

```
(M >0).any()
# retourne True si au
#moins un élément de M
```

```
M=np.array([1,-2,5,9,51])
x=(M >0).any()
print(x)
```

Le script affiche :

```
True
```

#satisfait la condition #booléenne		

x=(M >0).all() # retourne True si tous les #éléments de M satisfont la #condition #booléenne .	M=np.array([1,-2,5,9,51]) x=(M >0).all() print(x)	Le script affiche : False

(M1 = = M2).all() #retourne True si les deux tableaux #sont égaux	M1=np.array([1,-2,5,9,51]) M2=np.array([1,-2,5,51,9]) x=(M1==M2).all() y=(M1==M2).any() print(x) print(y)	Le script affiche : False True

I-10. Sous-module linalg

On peut utiliser le sous module linalg pour faire des calculs vectorisés :

- **np.linalg.det(M)** calcule le déterminant de M
- **np.linalg.inv(M)** calcule l'inverse de la matrice carrée M
- **np.linalg.matrix_rank(M)** calcule le rang de la matrice M
- **np.linalg.solve(a,b)** résout le système linéaire ax = b où a est une matrice carrée inversible

Exemple :

```
a = np.array([[1/(i+j+2) for i in range(4)] for j in range(4)])
print('a : \n',a)
print('det : ', np.linalg.det(a))
print('inv : \n', np.linalg.inv(a))
b= np.array(np.arange(4))
y=np.linalg.solve(a,b)
print('la solution est  : ',y)
```

Le script affiche :

```
a :
 [[ 0.5        0.33333333  0.25        0.2       ]
 [ 0.33333333  0.25        0.2         0.16666667]
 [ 0.25        0.2         0.16666667  0.14285714]
 [ 0.2         0.16666667  0.14285714  0.125     ]]
det :  2.36205593348e-09
inv :
 [[  200.         -1200.          2100.         -1120.        ]
 [ -1200.          8100.        -15120.00000001   8400.00000001]
 [  2100.        -15120.00000001  29400.00000002 -16800.00000001]
 [ -1120.          8400.00000001 -16800.00000001   9800.00000001]]
la solution est  :  [ -360.  3060. -6720.  4200.]
```

II. Graphiques avec le module matplotlib:

II-1. Importer le module matplotlib et le sous module pyplot

```
import matplotlib.pyplot as plt
```

II-2. Commandes utiles de Matplotlib

a) plot, show, savefig et clf

- **plt.plot(x,y):** permet de représenter des fonctions en reliant par une ligne les couples de points (x[i],y[i]) successifs contenus dans les deux tableaux x et y (qui doivent donc être de même taille).
  La commande admet une multitude d'options,

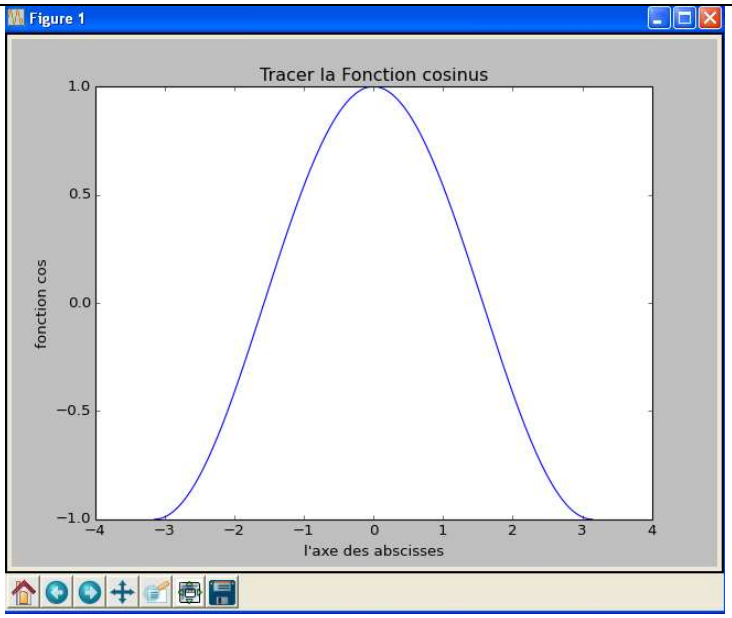
- **plt.show():** déclenche l'affichage à l'écran.
  Attention, cela stoppe l'exécution du script en cours jusqu'à ce que vous ayez fermé la fenêtre d'affichage.

- **plt.savefig(nomfichier):** permet de sauvegarder la figure en cours dans le fichier nomfichier.

- **plt.clf():** efface la figure en cours (clf pour « CLear Figure ») car plt continue d'ajouter des éléments à la même figure tant qu'on ne lui a pas dit d'arrêter.

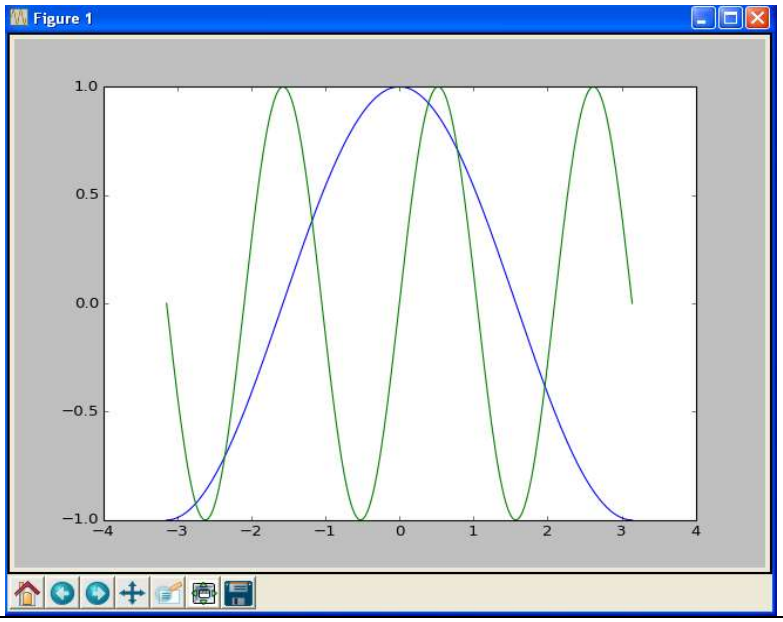
Exemple 1 :

```
import numpy as np
import matplotlib.pyplot as plt
# Un dessin très simple commence toujours par un échantillonnage de l'axe des x
X = np.linspace(-np.pi,np.pi,256)
# Les fonctions de numpy peuvent s'appliquer séquentiellement à tous les éléments d'un tableau
#numpy:
Y = np.cos(X)
plt.plot(X,Y) # Maintenant, il faut demander l'affichage de la courbe
plt.ylabel("fonction cos")
plt.xlabel("l'axe des abscisses")
plt.title("Tracer la Fonction cosinus") #L'ajout d'un titre
#l'affichage du graphe sur l'écran
plt.show()  #ou plt.savefig('cos.pdf') si on veut sauvegarder le graphe sous forme d'un fichier
plt.clf() # On nettoie la figure si jamais on veut en faire d'autres par après
```



Exemple 2 : plusieurs courbes superposées

```
import numpy as np
import matplotlib.pyplot as plt
X = np.linspace(-np.pi,np.pi,256)
Y = np.cos(X)
Z = np.sin(3*X)
# Maintenant, il faut demander l'affichage des courbes (superposées par défaut)
plt.plot(X,Y)
plt.plot(X,Z)
plt.show()#ou plt.savefig('cos_sin.pdf')
plt.clf()
```

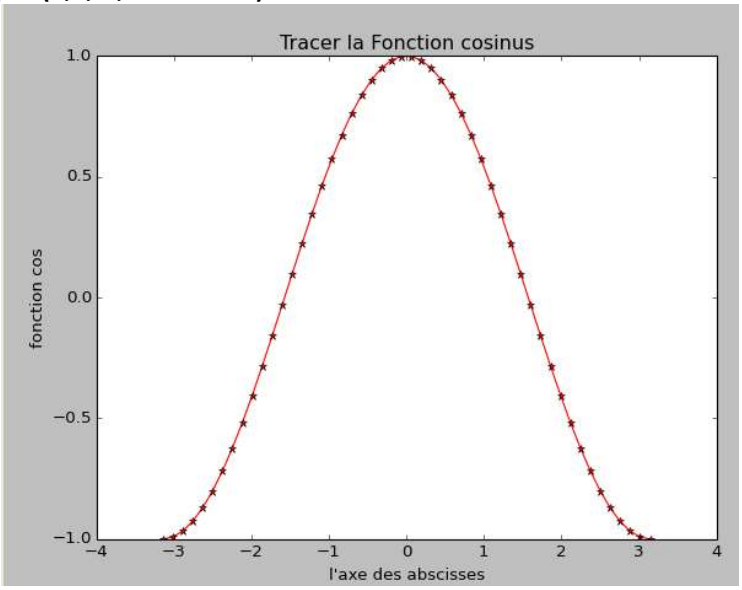


Pour connaître toutes les options de **plt.plot**, le mieux est de se référer à la documentation de Matplotlib (**help(plt.plot)**) . Voyons ici quelques unes d'entre elles :

- **couleur du trait** : On peut choisir la couleur de la courbe en mettant une lettre de couleur entre guillemets :
  exemple : **plt.plot(X,Y,'r') #** tracera notre courbe en rouge.

valeur	correspondance
r	rouge
g	vert
c	cyan
m	magenta
y	jaune
k	noir
w	blanc

- **symboles** : mettre des symboles aux points tracés se fait via l'option **marker**. Les possibilités sont nombreuses parmi **['+' | '*' | ',' | '.' | '1' | '2' | '3' | '4' | '<' | '>' | 'D' | 'H' | '^' | '_' | 'd' | 'h' | 'o' | 'p' | 's' | 'v' | 'x' | '|' | TICKUP | TICKDOWN | TICKLEFT | TICKRIGHT | 'None' | ' ' | ''].**
  exemple : **plt.plot(X,Y,'r',marker=' +')**



- **style du trait** : pointillés, absences de trait, etc se décident avec l'option **ls**.
  ls='-' ligne continue, s= '--' tirets, s='-.' points-tirets, s=':' pointillés
  exemple : **plt.plot(X,Y,'k',ls=':') #** tracera notre courbe en noire avec des pointillés.

- **épaisseur du trait** : **lw**=flottant (comme linewidth=2) donne un trait, pointillé (tout ce qui est défini par style du trait) d'épaiseur "flottant" en points.
  exemple : **plt.plot(X,Y,'k',ls='-',lw=10)**